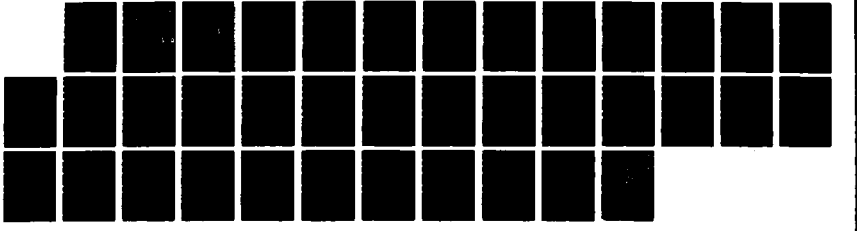
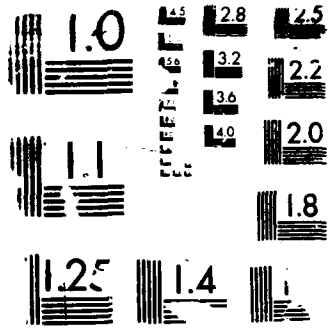
AD-A189 569 WHY WE CAN'T PROGRAM MULTIPROCESSORS THE WAY WE'RE TRYING TO DO IT NOW(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE D BALDWIN AUG 87 TR-224 1/1

UNCLASSIFIED DACA76-85-C-0001 F/G 12/5 NL





RESOLUTION TEST CHART

AD-A189 569

DTIC FILE COPY

Why We Can't Program Multiprocessors the Way We're Trying to Do It Now

Doug Baldwin
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 224
August 1987

DTIC
ELECTE
JAN 15 1988
S D
H

Department of Computer Science
University of Rochester
Rochester, New York 14627

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

87 12 22 010

Why We Can't Program Multiprocessors the Way We're Trying to Do It Now

Doug Baldwin
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 224
August 1987

DTIC
ELECTE
JAN 1 5 1988
S D
H

Abstract

It is now rather easy to build the hardware of a multiprocessor computer, but still quite difficult to program it to do useful work. I argue that a key cause of this problem is that the models of computation on which current programming languages are based are inadequate for describing parallelism. Programming languages are classified according to their underlying models of computation. The two major classifications are imperative languages, based on machine-oriented models, and declarative languages, based on mathematical abstractions. Imperative languages rely on side-effects to advance computations, and so are inherently sequential. Declarative languages lack general ways of describing data parallelism, and so can only express a fraction of the potential parallelism in a program. The paper closes with a brief description of current work on constraint languages, which might help to reduce the problems of programming multiprocessors.

DTIC COPY INSPECTED 6

The research described in this paper was funded by the National Science Foundation under Grants DMC-8613489 and DCR-8320136 and by the U.S. Army Engineering Topographic Laboratories under Contract DACA76-85-C-0001. The opinions expressed in this paper are those of the author, and are not necessarily endorsed by the above agencies.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

on For
RA&I
B
nced
cation
tion/
Availability Codes
Avail and/or
Dist Special
A-1

Contents

1. Introduction 1

1.1 Parallel Computing 1

1.2 Programming Languages 4

2. How Current Languages Limit Parallelism 7

2.1 Data Dependencies 7

2.2 Data Parallelism 11

2.3 Granularity 15

2.4 Generality 16

3. A Step Towards a Solution? 17

4. Summary and Conclusions 18

I. Implementations of "Totals" 19

1 Introduction

Parallel computation is an area in which software technology lags considerably behind hardware technology. The need for parallel computing in a number of applications (e.g., scientific computing, machine vision, artificial intelligence) is unquestioned, and computers with hundreds of processors are now readily available (for instance, the Butterfly™ [3] or the many derivatives of the Cosmic Cube [37]). However, these machines are programmed in essentially the same way as existing sequential machines. The best available parallel programming languages are variants of standard sequential languages, with extensions to let the programmer explicitly divide a program into tasks and pass information between those tasks. Although designers of these languages claim that they are no harder to use than conventional sequential ones [16, 18], programmers still face the problem of figuring out how to partition their application into tasks in addition to the usual problem of translating it into a program. An appealing alternative is to leave partitioning of programs to compilers. By hiding partitioning problems from programmers, this approach should make multi-processor computers easier to program than they are now. Unfortunately efforts to develop parallelizing compilers have so far been rather unsuccessful. Few have been used outside of laboratory settings, and most are restricted to specific programming styles or target architectures. In this paper I argue that the models of computation on which current programming languages are based are inadequate for expressing significant amounts of parallelism. This means that extending existing languages so that programmers can partition programs will be an awkward job at best, and that source languages are an important factor limiting the success of parallelizing compilers. Research to identify more appropriate languages for parallel programming is needed before multiprocessors will be widely useful.

1.1 Parallel Computing

For the purposes of this paper "parallel computing" means *general purpose* parallel computing. In other words, I do not view parallelism as a tool for only the areas that have traditionally exploited it, but rather as one that can be useful in arbitrary programs. I therefore adopt a MIMD (multiple instruction stream, multiple data stream) model of parallel computing, i.e., a model in which a single job is distributed over a number of processors, each able to execute its own code on arbitrary data independent of what the others are doing. Since it imposes no restrictions on how a job can be partitioned into concurrent parts, this model supports the widest possible variety of styles and uses of parallel programming. Communication between processors may be through shared memory, networking, or any other mechanism. The only requirement is that interprocessor communication be cheap enough that some jobs are worth partitioning into multiple cooperating processes. Throughout this paper I use the term *multiprocessor* to denote a computer system that implements this MIMD model.

"Butterfly" is a trademark of BBN Laboratories, Inc.

```
(1)  Record Format:
(2)      Type: An integer between 1 and 5 indicating the record's type.
(3)      Val:  A real number representing the principal value of the record.
(4)      Aux:  An array of real-valued fields containing auxiliary information.
(5)  Algorithm "Totals":
(6)      Get initial values for each total from user.
(7)      Initialise each count to 0.
(8)      Process each record as follows:
(9)          Increment the counter for this record's type by 1.
(10)         If the record is of type 1:
(11)           Add the record's val field to the type 1 total.
(12)         If the record is of type 2:
(13)           If the record's val field is an integer multiple of 3
(14)             Apply function f to the record's val field,
(15)           Otherwise
(16)             Apply function g to the record's val field.
(17)           Add the result of the chosen function application to the type 2 total.
(18)         If the record is of type 3:
(19)           Add each odd-indexed aux field to the type 3 total.
(20)           Set each even-indexed aux field to the succeeding even-indexed
(21)               aux field divided by 2.
(22)         If the record is of type 4:
(23)           Add the natural logarithm of the record's val field to the type 4 total.
(24)         If the record is of type 5:
(25)           Add the sum of f applied to the record's val field and
(26)               g applied to the record's val field to the type 5 total.
```

Figure 1. "Totals": A Demonstration of Parallelism

The process of breaking a program into processes that can run concurrently on a multiprocessor is called *parallelization*. Parallelization can (in principle) be done by programmers, by compilers, or by both working together. It is sometimes convenient to think of the result of parallelization as a schedule that indicates the order in which parts of a program are executed. Parallelization is assumed to produce a schedule in which some operations really do execute simultaneously (in other words, the trivial schedule in which everything is done sequentially is not parallelized). Parallelization can be done at many different levels of *granularity*, determining the size of the resulting processes. For example, very fine granularity might yield processes corresponding to single machine instructions, whereas a coarser (and more realistic) granularity might yield processes corresponding to procedures or other large blocks of source code. The most appropriate granularity depends on who or what is doing the parallelization and on the target machine's architecture.

As an example of some of the ways parallelism may appear in an algorithm, consider "Totals", the algorithm shown in Figure 1. The actual computation done by this algorithm is nonsense; its only purpose is to demonstrate parallelism in a number of settings. "Totals" works on a set of records, with each record having one of five types. The purpose of the algorithm is to count the number of records of each type and total up the records within each type. Unfortunately the total for each type must be computed in an idiosyncratic way. Furthermore, each total starts at a user-defined value rather than at 0. The functions f and g called by "Totals" are assumed to be real-valued, but are otherwise left to the reader's imagination.

The most significant kind of parallelism in "Totals" is *data parallelism*, the possibility of processing a number of different data items concurrently. The entire

algorithm can be expressed in two different data parallel forms: In the first, each input record is assigned to a distinct process, which checks the record's type and updates the appropriate total and count. In the second, each total or count is computed by a distinct process, which extracts from the input those records whose type corresponds to the result it is computing. In other words, the parallelism can be either over inputs to "Totals" or over outputs. If the data parallelism is over inputs, then there will probably be a large number of processes acting at once, but they will need to synchronize their accesses to the totals and counts. Parallelism over outputs avoids the need for this synchronization, but involves fewer processes (and so possibly more time to complete the overall computation). Needless to say, other algorithms would exhibit different trade-offs between parallelism over inputs and parallelism over outputs. Note that general data parallelism does not require all items to be processed in the same way. Adding such a restriction yields a form of data parallelism often called *vector parallelism*. An example can be seen in lines 20 and 21 of Figure 1, where with suitable synchronization of reads and updates, the new values of all even-indexed *aux* fields can be computed at once.

"Totals" demonstrates a number of forms of parallelism other than data parallelism, including the following:

- The initializations in lines 6 and 7 can be done in parallel, i.e., the counts can be set to zero while initial totals are being read from the user. Similarly, the count and total for each record's type could be updated in parallel (i.e., line 9 could be done in parallel with lines 10 through 26). This sort of statement- or block-level parallelism is typical of that sought by most parallelizing compilers.
- The conditional in lines 13 through 16 could be parallelized by having one process decide whether the current record's *val* field is a multiple of 3 while two others apply f and g to that field, saving results in distinct temporaries. One of these results can then be added to the type 2 total, depending on the result of the test. The same idea could be applied to dispatching on record types (lines 10, 12, 18, 22, and 24). This kind of parallelism is related to the "or parallelism" that certain compilers for logic languages try to exploit.
- Lines 25 and 26 apply f and g and then add the results together. Assuming that f and g do not interfere with each other (for instance by changing shared data), the two applications can be done in parallel. This is an example of *argument parallelism* (the arguments to the addition are evaluated in parallel), exploited by parallelizing compilers for functional languages.
- Line 19 involves summing a (possibly) large number of *aux* fields. Since addition is associative, this summation can be broken into a number of smaller partial sums that can be done in parallel.
- Line 23 demonstrates a case in which a complex operation (taking a natural logarithm) is treated as primitive, in the sense that no further description is given of how the operation is to be carried out. Nonetheless, one can imagine exploiting parallelism within such operations, perhaps by providing parallelized library functions to implement them.

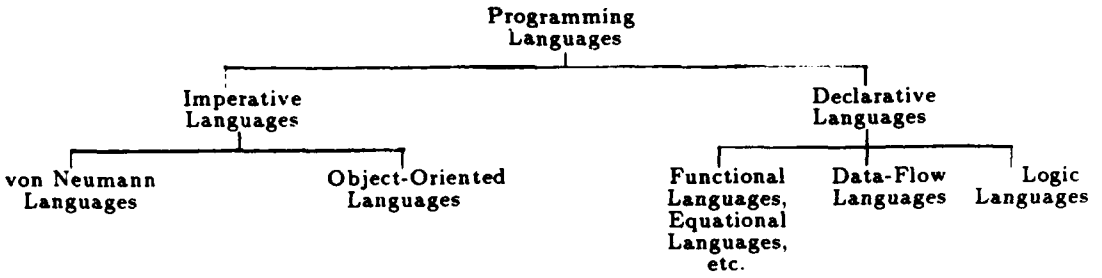


Figure 2: A Taxonomy of Programming Languages

Implementations of "Totals" in a number of real programming languages are given in Appendix I. These implementations will be used throughout this paper as illustrations of the ways in which different kinds of language help or hinder the expression of parallelism.

1.2 Programming Languages

The basic thesis of this paper is that the models of programming embodied in currently used programming languages are inadequate for writing parallel programs. Defending such a sweeping statement requires some organized way of characterizing and discussing programming languages. The language classification used in this paper is shown in Figure 2. The most important distinction in this taxonomy is the one between imperative and declarative languages. A program in an imperative language explicitly describes how some (possibly abstract) computer changes its own state to produce a desired result; a declarative program is a mathematical description of the result itself (rather than of the process that produces it). Imperative languages are further sub-divided into the "von Neumann" languages and the object-oriented languages. The von Neumann languages closely reflect the von Neumann computer architecture, specifically by representing program state as a single, relatively unstructured set of variables (corresponding to the single, unstructured memory of the von Neumann architecture). Object-oriented languages compartmentalize program state and the operations that change it, but still reflect a very mechanical view of computation. Different sub-divisions of the declarative languages represent different mathematical formalisms, for example functional languages, logic languages, et cetera. The leaf classes from Figure 2 are described more carefully in the following paragraphs. Every class includes languages in which parallelism can be described explicitly (referred to later as "explicitly parallel" languages), as well as languages in which parallelism is at best implicit (sequential languages). As will be shown, even explicitly parallel languages impose the same barriers to parallel programming as do the sequential members of the same class.

von Neumann Languages. Most programming languages are von Neumann languages. As an example, Figure 4 (in Appendix I) shows "Totals" written in Pascal [24], a typical von Neumann language. Von Neumann languages are fundamentally sequential, since the order in which program states are entered influences the result produced. Nonetheless, it is possible to extend the von Neumann model to allow

explicit parallelism, and this has been done. For example, Hoare's Communicating Sequential Processes [23] models parallel computation as a number of von Neumann processes passing data between themselves over special connections. The assumption is that the only thing that von Neumann languages need in order to express parallelism is the ability to describe communication between processes. Another way of extending the von Neumann model to parallel computation is reflected in Linda [19]. Linda tries to avoid commitment to a particular computational model for individual processes, but does assume that all communication between processes is done by depositing information into and extracting it from a global, unstructured, "tuple space". In other words, Linda is based on the assumption that the von Neumann approach to maintaining state can be applied to the global state of a parallel program.[1]

Object-oriented languages. Object-oriented languages are imperative languages in which the set of variables defining a program's state is partitioned into small subsets. Each subset of variables is encapsulated along with the procedures that access its members in an *object*. Objects interact by sending *messages* to each other. A message causes one of its recipient's procedures (also called a *method* for the message) to be executed, presumably changing or reporting the values of some of the recipient's state variables. A typical object-oriented program uses many *instances* of each type (or *class*) of object, i.e., has many objects with the same procedures but distinct state variables. The best current example of an object-oriented language is Smalltalk [20]. An implementation of "Totals" in Smalltalk appears in Figures 5a, 5b, and 5c (in Appendix I).

There is an obvious correspondence between objects passing messages between themselves and processes passing messages between themselves, and so object-oriented programming seems to be a natural way to explicitly describe parallelism. Emerald [7] is a recent example of an object-oriented[2] language designed for parallel programming. Much more common than the use of fully object-oriented languages for parallel programming is the use of an object-like model of processes in an otherwise von Neumann language. Processes in these languages act like objects in that they contain private state information that they manipulate in response to messages from other processes. These languages do not, however, extend the object model to data and procedures within individual processes. Examples of object-like processes include guardians in Argus [27] and processes in SR [2] or Lynx [36]. Figures 6a through 6g in Appendix I show an implementation of "Totals" in Lynx as an example of this kind of language.

---

[1] Linda's tuple space differs from traditional von Neumann memories by being associatively addressed. For the purposes of this paper however, the crucial similarity is that in both cases communication between parts of a program is achieved by making long-lived changes to the state of tuple space or memory.

[2] Purists might prefer "object-based", since Emerald lacks features that are sometimes considered important for object-oriented programming.

Functional languages. Functional languages, also known as applicative languages, are declarative languages that use functions as the underlying mathematical abstraction [4]. In other words, programs are viewed as pure functions, generally composed out of simpler functions. Functional languages have an important implicit source of parallelism, namely that all the arguments of a function application can be evaluated concurrently (argument parallelism). Most existing functional languages are still laboratory prototypes, so there are no widely used examples of the class. The best example is probably Lisp, which has an applicative subset ("pure Lisp"), although all Lisp dialects also include non-applicative features. Figure 8 (see Appendix I) gives an implementation of "Totals" in an applicative subset of Common Lisp [40].

Data-flow languages. Data-flow languages [1] were deliberately developed to describe parallelism in computation. A data-flow program is essentially a description of a directed graph, in which nodes represent functions and an edge from node a to node b means that the output of the function represented by a is an input to the function represented by b. The edges of this graph describe the serializing dependencies between functions: no function application can be evaluated before all of its inputs have been computed, but any applications not thus dependent on one another may be simultaneous. Data-flow languages are very closely related to functional languages, in that both view computation as function application and composition. However, data-flow languages generally provide a richer set of constructs for describing the structure of the dependency graph than do functional languages, and so may support more ways of describing parallelism. This point is discussed in more detail later. Like functional languages, data-flow languages are still mainly experimental — the description of VAL [29] is probably the most accessible example of the class.

Logic languages. Logic languages are declarative languages in which programs are sets of relations between objects. These relations are stated either as facts ("r holds between a and b"), or as if-then proof rules ("r holds if s holds and t holds and ..."). The best examples of logic languages are Prolog [12] and its variants. Execution of a logic program can be viewed as an attempt to prove that some specific relation holds between certain objects. In principle, correct execution does not depend on either the order in which sub-goals arising in the proof are proved or the order in which alternative proofs are tried. Thus there is potentially a great deal of parallelism implicit in a logic program. Unfortunately, Prolog is defined in ways that make this parallelism very hard to detect automatically. A number of parallel variants of Prolog have been proposed in which parallelism is easier to detect [11, 38]. These variants are semi-explicitly parallel, in that programmers must make certain aspects of parallel execution explicit in their programs, whereas other aspects can be determined automatically.

2 How Current Languages Limit Parallelism

From the preceding survey of programming languages it might seem that parallelism is supported in many ways by virtually every language. However, it turns out that this apparent support is insufficient for general-purpose parallel programming. The reason is that there are several important sources of and limits to parallelism that cannot be expressed clearly in any existing language. This problem is obviously serious for sequential languages, since they must be translated into parallel form explicitly. Without clear guidance from the source text as to what parallelizations are legal, and more importantly what ones are not legal, it will be very hard to produce correct and efficient parallelizations of sequential code. These comments apply equally to automatic and manual translation. The symptom of the problem for explicitly parallel languages is similar: it is hard to tell whether or not the parallelization given explicitly in a program is actually correct, i.e., whether it is consistent with the basic computation desired of the program. What is missing is a clear statement of the parallelism *potentially* allowed by this basic computation — essentially the same thing as was missing from sequential languages. Thus, although explicitly parallel languages may encourage the problem to show up during debugging rather than during coding, both they and sequential languages suffer from the same thing: inability to clearly express potential parallelism. The rest of this section discusses the basic reasons why parallelism is hard to express, independent of whether that parallelism is implicit or explicit.

2.1 Data Dependencies

The key problem in writing parallel programs is that of describing *data dependencies*, i.e., indicating when two operations must be done sequentially because one produces or destroys a value that the other needs. Generally, any operations that are not explicitly serialized by data dependencies can be done in parallel. Conversely, any operations between which there is a data dependency must be done sequentially. The ultimate goal in designing a parallel programming language should be to allow those data dependencies that are essential to the algorithm being programmed to be stated clearly, while eliminating any need for extraneous dependencies.

Data dependencies are an intimate part of imperative languages. Recall that imperative languages are characterized by a model of programming in which each statement contributes to solving a problem by making some change to the state of an abstract computing machine. These changes in state are called *side effects*. Side effects are relatively permanent, in that they can be visible arbitrarily long after the statement that caused them finishes executing. In fact, since statement execution is a transient activity, side effects are the *only* way an imperative program makes permanent progress towards completion. Note that the result of executing a statement in an imperative language generally depends on the machine state (i.e., previously caused side effects) immediately prior to its execution. Thus the relative order in which any pair of statements is executed can be crucial to the outcome of a program. Some of the resulting data dependencies are fairly easy to see, for

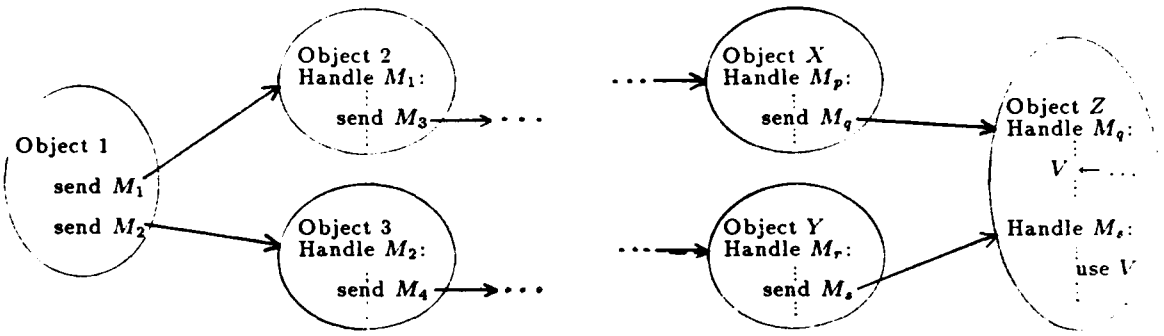


Figure 3. Indirect Data Dependencies in Object-Oriented Languages

example because they involve the same variables appearing in different statements. Others are much less obvious, for example because they involve an implicit "program counter" or other control mechanism (such dependencies are often called *control dependencies*). In short, imperative languages are inherently sequential, being very deeply founded on the assumption that any two statements will be executed in some definite order relative to one another. Violations of this assumption can only be justified by proving that none of the dependencies between the statements in question can influence the result of the computation. One of the consequences for parallel programming is that more effort is often devoted to properly starting processes, synchronizing them, and passing data between them than is devoted to coding the basic algorithm underlying a program. Compare the lengths of the Lynx and Pascal versions of "Totals" in Appendix I for an example of this effect.

Data dependencies are very difficult to detect in imperative languages. One of the reasons is that side effects do not necessarily have any locality. In other words, a side effect can be observed arbitrarily long after it is caused, by code arbitrarily far removed from the causer. Object-oriented languages and von Neumann languages that support modular programs might seem to alleviate this problem by grouping data and the procedures that manipulate them together, but this alleviation is largely illusory. The problem is that a message might have side effects that affect the way the receiving object handles later messages. Similarly, calling a subprogram located in some module may have side effects that influence the behavior of later calls to other subprograms in the same module. Because an object's handling of a message may involve sending other messages to other objects, and a subprogram may call other subprograms in other modules, identifying the final side effects of a message or call can require the same global analysis in highly modular languages as in languages with no support for modularity. If anything, the analysis may be harder in object-oriented or modular languages because it has to cross supposedly inviolable object or module boundaries. Figure 3 illustrates the problem for object-oriented languages.

The worst problems with data dependencies in imperative languages stem from *aliasing*. Aliasing occurs whenever a single datum may have several different names. Common examples include array references ($a[i]$ and $a[j]$ name the same object if $i = j$) and references through pointers. A good "real life" example appears on line

36 of Figure 4 in the Appendix, where "$aux[j-1]$" on one iteration of the loop refers to the item named by "$aux[j+1]$" on the previous iteration, even though the two names are distinct. Notice how understanding this example requires understanding the behavior of the loop in which it is embedded. The need for this kind of fairly deep understanding of the context in which aliases occur is typical of the problems of alias detection. Aliasing means that data dependencies cannot be detected simply by looking for occurrences of common variable names in different statements — a data dependency may exist between two statements even if they have no names in common if the shared data are named by different aliases in each statement. Techniques have been developed for recognizing aliases involving some forms of array reference [6, 31], but they do not generalize to other sources of aliasing. In fact, the general problem of deciding whether two names are aliases for each other is undecidable.[3] Unfortunately, recognizing aliases is very important for extracting parallelism from imperative programs. The situations in which aliasing is most likely to occur are precisely those in which vast amounts of parallelism are also likely to occur — for example, large arrays or structures linked together by pointers, where it might be possible to process all elements simultaneously. However, it is generally impossible to prove that this concurrency respects data dependencies without being able to tell whether apparently distinct references to the same structure are aliases for each other. The impossibility of general alias recognition means that the quality of automatic parallelizations of imperative programs will always be limited. Furthermore, the restricted domains of existing alias recognizers suggest that these limits will be quite severe, at least for the foreseeable future. The problem is less severe for manual parallelization, since programmers can be expected to understand aliasing in their programs better than a compiler would. Nonetheless, even programmers seldom understand their programs perfectly. Bugs in parallel programs can be extremely difficult to detect and fix, and the problem is compounded if the bug is disguised by aliasing. Thus aliasing not only makes parallelization of sequential imperative programs impossible, it is also a strong reason to suspect that explicitly parallel imperative programs will be harder to write and less reliable than traditional sequential ones.

Despite the problems described above, there has been a considerable amount of work on automatically finding parallelism in sequential imperative languages. This work has met with a certain amount of success, for example the Bulldog compiler [14] or any of a number of vectorizing compilers for array processors and supercomputers [10, 33]. A large catalogue of techniques for detecting and exploiting parallelism in imperative languages is given in [32]. All of these systems, however, are successful only for restricted kinds of program and target machine. For example, none can handle parallelism involving dynamic data structures, and even

[3] For example, if an alias detector existed then a program of the form "if $a[i]$ and $a[j]$ are aliases at the end of this program, then make i and j unequal, otherwise make them equal" could be written to achieve the paradox of having two names be aliases if and only if they weren't aliases.

array accesses can only be parallelized when subscript expressions take simple forms that allow aliases to be recognized. Many common constructs, for example "while" loops or subprogram calls, cannot be parallelized by these systems. Some compilers (Bulldog for instance) are based on assumptions about the target machine that simplify or even eliminate problems of synchronizing concurrent tasks. Often the kinds of parallelism detected are also very limited. For example, vectorizing compilers usually ignore all sources of parallelism except vector parallelism arising from very simple loops (typically only inner-most loops, loops not containing subprogram calls or conditionals, et cetera). Although it is true that programs from some large and important classes (notably computation-intensive numeric programs) have a sufficiently regular structure that these techniques can find substantial parallelism, the techniques are clearly not general purpose. Furthermore, many of the limitations are deep consequences of the fundamental problems discussed above (especially alias detection), and so will not be easy to remove. Thus, while it is true that parallelism in imperative languages can sometimes be exploited automatically, doing it in general is still impossible.

Data dependency detection is a problem mainly for imperative languages. The fact that these languages are built on a side-effect based model of computation makes data dependency detection both extremely important for parallelization and extremely hard to do. The situation is much better in declarative languages. The mathematical formalisms on which these languages are based generally disallow side-effects. Thus a variable's value is defined in exactly one place, and is never changed after being defined. This "single assignment" property means that the only dependencies that can arise are between an object's definition and its uses. Thus the reason for analyzing data dependencies in declarative languages is to answer the question "does this object have a value yet?", whereas with imperative languages the question is "of all the values that this object has at various times, does it now have the right one for this use?" This difference in emphasis means that adequate dependency analysis can be done with much less information than in imperative languages. For example, if it is known that the array a is fully defined at some point in a program, then operations referencing $a[i]$ and $a[j]$ can always be done in parallel, regardless of whether $a[i]$ and $a[j]$ may be aliases for each other. It is still impossible to do a perfect analysis at compile time (for instance, if a were only *partially* defined in the above example, possible aliasing might prevent parallelization), but dependency detection can also be deferred until run time. A simple way of doing this is simply to reserve a special "not yet defined" value or flag for each data object, with users of an object suspending themselves until the object has been defined. One can imagine similar run time dependency detection for imperative languages, and at least one multiprocessor has been designed with hardware support for some of the necessary synchronization of data accesses [39]. However, the greater amount of information required means that recognizing data dependencies for imperative programs involves more overhead than doing it for declarative programs. Even with hardware assistance, this overhead is generally unacceptable if incurred on every run of a program.

2.2 Data Parallelism

Intuitively, data parallelism is an important factor in truly massive parallelizations because programs commonly process data sets consisting of hundreds or thousands of items. If these items can be processed independently of one another, then speed-up factors of hundreds or thousands can be obtained (under ideal circumstances, of course). Another argument for the importance of data parallelism, paraphrased from [22], is as follows: A given program contains only a fixed number of instructions, which necessarily limits the benefit to be gained from simply rearranging those instructions into some parallel schedule. On the other hand, the same program can typically be applied to arbitrarily large data sets, meaning that the benefit to be gained from data parallelism is potentially unbounded. Finally, data parallelism has a multiplicative effect on other sources of parallelism. For example, if the body of a loop can be sped up by a factor of 5 due to various forms of control parallelism, and the loop iterates over 20 items, then replicating the parallelized loop to exploit data parallelism ideally gives a net speed-up of 100. Suggestive experimental evidence for the importance of data parallelism can be found in Ellis's measurements of speed-ups produced by the Bulldog compiler [14]. One of the factors considered in these experiments was the extent to which replicating loop bodies (loop unrolling) affected the amount of parallelism detected by the compiler. It was quite common for programs in which loop bodies had been replicated 16 times to exhibit 10 times more parallelism on the largest data sets tested than the same programs without unrolling.[4] Although Ellis was not specifically looking for data parallelism, the fact that typical loops process a distinct datum on each iteration makes it a safe guess that most of the differences he observed were due to changes in the amount of data parallelism that was exploited. This guess is supported by the fact that the amount of parallelism that was detected in programs with large amounts of loop unrolling was much more sensitive to data set size than was the case with the same programs without unrolling. All in all, easy expression and detection of data parallelism appears to be a very important feature for a parallel programming language.

In imperative languages, potentially data parallel computations are usually written as a loop over the appropriate data structures. This is one of the hardest forms in which to detect parallelism automatically. First, whether parallelization is possible or not depends very much on how independent each iteration is of the others, determining which can require very precise information about aliasing. Even with accurate information about aliasing, analysis of data dependencies in potentially data parallel loops necessarily crosses basic block[5] boundaries — even in the best case the body of a loop is a single basic block, and the analysis must cross the boundaries of this block when considering dependencies between two iterations.

[4] One might expect a factor of 16 improvement; the difference is presumably due to iterations not being perfectly independent of each other.

[5] A basic block is a straight-line sequence of code with one entry point and one exit point.

dependencies between the first iteration and initialization code, et cetera. Data dependency analysis across basic block boundaries is not impossible (for instance, see [15]), but it is much harder than analysis within basic blocks. For example, when basic block boundaries are crossed uses of variables can be dependent on multiple definitions and vice versa, it is impossible to tell whether some of these dependencies only arise in special situations that can be avoided by using various run-time tricks, et cetera.

Looping in imperative languages generally corresponds to recursion in declarative ones. Thus the most general way of expressing a data parallel computation in a declarative language is through a recursive definition of some sort. For example, note how the main loop in "Totals" is implemented in Lisp or Prolog (function "Total" in Figure 8 and predicate "totals" in Figure 7 from Appendix I). Unfortunately, in all current declarative languages, these recursive definitions introduce apparent data dependencies involving parameters to or results from the recursion. Often the dependencies are spurious, in that the order they impose on the computation is irrelevant to its correctness. Distinguishing spurious data dependencies from vital ones, however, requires deep knowledge of the algorithm in which they appear, and it is unrealistic to expect a compiler to have such knowledge. Implementations of current declarative languages must therefore carry out recursive computations sequentially, even those that actually contain substantial data parallelism. Declarative languages thus give up a very important source of parallelism, meaning that they can only express a fraction of the parallelism present in a program in a practically usable form.

There are a number of ways of reducing the problems associated with data parallelism in declarative languages. Some are specific to certain kinds of declarative language, others are more general. The most obvious general approach is to organize programs in ways that make data parallelism appear as some more easily recognized kind of parallelism (typically argument parallelism). For example, consider the problem of initializing a set of n data items. In theory, each item can be initialized in parallel with all others, achieving an n-fold speed-up. In a naive declarative implementation the set might be represented as a list, and the initialization done somewhat as follows (using pseudo-Lisp as a paradigmatic declarative language):

```
(define Init ()
   (if (enough-initialized)
      nil
      (cons (One-Initialized-Datum) (Init))))
```

Essentially no parallelization is possible here, because data are produced one at a time and there is a data dependency between the result returned by each invocation of "Init" and the result returned by its caller. Alternatively, the set could be organized as a binary tree, in which case initialization could be done as

```
(define Init ()
   (if (deep-enough)
      (One-Initialized-Datum)
      (make-tree-node (Init) (Init))))
```

Here each invocation of "Init" causes two more recursive invocations, apparently as parallel arguments to "make-tree-node". It is fairly simple to exploit this parallelism in order to get an $\frac{n}{\log n}$ speed-up. This improvement can be quite dramatic, even if not as good as full data parallelism. Thus careful coding can allow some data parallelism to be exploited, but it is not a complete solution to the problem for several reasons. First, it does not always make the maximum possible amount of data parallelism visible; second, it requires programmers to be aware of both the parallelism in their applications and the ways in which their language implementation can exploit it.

Functional languages provide another partial solution to the data parallelism problem with *mapping functions*. A mapping function is a function that explicitly applies a second function to all elements of a list or other data structure. Any use of a mapping function can obviously be compiled into data parallel object code, as long as the mapping function is not defined as processing the elements of the data structure in some fixed order. However, mapping functions are only available for data structures that are primitives of a language, not for structures defined by users. Thus there are likely to be many cases in which data parallelism is available, but not in a form that can easily be expressed using a language's mapping functions. Another problem with mapping functions is that data parallelism is sometimes over the outputs of a computation rather than over its inputs (see the discussion of "Totals" in the introduction for an example). Because functions are directional (i.e., have distinct inputs and outputs), mapping functions cannot be used to map a data parallel computation over an as-yet-unknown set of output values. Finally, many data parallel computations produce some aggregate of their inputs as a result (for example the sum of the elements in an array). Because of the single assignment property, instances of a mapped function cannot share and jointly update such aggregates. Thus mapping functions do not address this kind of data parallelism at all. It is also worth noting that the idea of mapping an operation over a data structure has been used to make data parallelism more or less explicit in several nondeclarative languages. Examples include Hillis's proposals for programming the Connection Machine [21], FORTRAN 8X [30], and APL [34]. All of these languages incorporate the disadvantages of mapping functions as well as the advantages.

Data flow languages are much like functional languages, since the basic computational elements of each are functions. However, data flow languages also include the idea of a directed "data flow graph" that describes how values are passed between outputs and inputs of functions. Data flow languages can include statements that define the structure of this graph in addition to statements that define the functions that make up its nodes. VAL's "forall" [29] can be viewed as an example of such a structure-defining statement. Although VAL does not seem to have aggressively pursued the idea of structure-defining statements, it should be easy to devise data flow languages that do. Explicit descriptions of data flow structure can support many forms of data parallelism, including parallelism over the outputs of a computation. However, it is not clear whether structure-defining statements that are applicable to arbitrary user-defined data structures can be devised, and

such statements do not really address the problem of aggregating the results of a data parallel computation (VAL provides for limited kinds of aggregation, but not in a general way). Thus, although data flow languages can express data parallelism better than functional languages, their potential in this area has not yet been fully developed and they do not appear able to handle all common forms of data parallelism.

Of all declarative languages, logic languages are best suited to describing data parallelism. One reason is that the main elements of a logic program are relations, not functions. Since relations do not distinguish inputs from outputs, it is no harder to describe data parallelism over "outputs" than over "inputs", simply because these terms have no real meaning to the language itself.

Another reason logic languages are relatively well-suited to expressing data parallelism is that they allow programmers to avoid unnecessary precision in programs. As discussed earlier, recursion is often used in declarative languages to describe data parallel computations. This recursion introduces data dependencies that seem to require the computation to be done in a specific order. The reason these dependencies induce a single order on the computation is that they imply that specific values are needed as inputs to specific operations. Often this specificity is misleading, in that any one of many values is equally permissible. For example, in totalling a set of numbers, the individual elements must be added to the total in some order, but the exact order is irrelevant. In such cases, needless precision in describing the order in which values are computed obscures possible parallelism (and not just data parallelism). Logic languages can reduce this problem by allowing programmers to write relations that can be satisfied by whole families of values. For example, here is a pseudo-Prolog program to total the numbers in a set:

```
total(S,0) :- empty(S).
total(S,T) :- member(X,S), remove(X,S,S2), total(S2,X2), T is X + X2.
```

Essentially this program says that the total of an empty set is 0; the total of a non-empty set is computed by removing one element from the set and then adding that element to the total of the resulting subset. Note that there is a chain of data dependencies that seems to imply that X must be identified and removed from S to yield S2 before the recursive totalling can be done. However, because the program does not specify *which* member of S is removed, just that *some* member is, there are $n!$ different orders in which an n element set can be totalled. A programmer could use this fact to prove various parallelizations of the totalling operation correct, and it is even conceivable that a sophisticated compiler could recognize that this calculation is largely order-independent and parallelize it accordingly. Unfortunately however, these uses of under-specification to support parallelism are tantalizing theoretical possibilities for which the technology that would allow automatic, or even semi-automatic, exploitation does not yet exist.

Note that in discussing data parallelism in logic languages, no mention was made of mapping operations or other explicit ways of indicating a potentially data parallel computation. Logic languages do not require such mechanisms, because

data parallelism is often an implicit consequence of the search strategies used to deduce conclusions from the rules and facts making up a program. Specifically, consider a typical data-parallel calculation in a logic program, which might have the form "for all elements of set A, prove that relation r holds". This expression is logically equivalent to "prove that there is no element of A for which r does not hold", which in turn is equivalent to "prove that the statement 'r does not hold of A_1 or r does not hold of A_2 or ...' is false", where A_1, A_2, et cetera represent the specific elements of A. This last form is just a large "or", the elements of which can in principle be checked in parallel. This kind of parallelism is called *or-parallelism* in the logic programming literature. Unfortunately, or-parallelism is hard to implement efficiently, mainly because the alternatives in an "or" may share global variables. Serious synchronization problems arise when the proofs of different alternatives involve binding different values to a shared variable. To summarize, logic languages provide several features that can be used to express data parallelism, but distinguishing parallel uses of these features from other, non-parallel, ones is very difficult.

2.3 Granularity

On most multiprocessors, passing data between processes and starting new ones are fairly expensive operations. Thus effectively using multiprocessors requires fairly coarse-grained parallelism in programs. In many parallel programming systems the "natural" granularity is too fine for use on multiprocessors. For example, data flow and functional languages are often accused of being unsuitable for multiprocessor parallelism because of this problem. In some cases granularity can be coarsened by collecting several fine-grained processes into a single aggregate process. This approach necessarily serializes the fine-grained operations within each aggregate, but if it also reduces the number of processes that need to be controlled and the total amount of interprocess communication it may still improve system performance. However, reducing interprocess communication requires that fine-grained processes be grouped into aggregates in such a way that most of the original communication ends up being between operations in the same aggregate. Reducing communication while retaining enough processes to provide significant parallelism requires that communication patterns in the original program be sparse (i.e., each original process communicates with only a few other processes). Unfortunately, fine-grained parallel programming systems do not always lead to sparse communication patterns.

One computational model that provides fine-grained parallelism but does not have sparse communication patterns is *connectionism* [35]. Connectionist models of computing are inspired by the brains of living organisms, in which sophisticated computations are carried out by networks of extremely simple neurons. Thus the basic computational element in a connectionist computing system is a cell whose output is determined by totalling and thresholding of inputs — different connectionist models differ in the set of values allowed as inputs and outputs, the ways in which inputs can be weighted within a cell, et cetera, but the key point is that

each cell performs a simple and fixed computation. The fine granularity of connectionist systems is a consequence of the simplicity of individual cells. Also because of the simplicity of cells, much of the power of connectionist systems comes from having very large numbers of cells with non-sparse interconnections between cells. Parallelism in connectionist systems comes from the large number of cells involved and the fact that all cells are active simultaneously. Because of the density of connections, efficient parallel execution of connectionist programs will probably require the development of specialized connectionist architectures rather than the use of general-purpose multiprocessors. It is also worth noting that only a very small amount of work has been done on designing programming languages based on connectionist models. AFL-1 [8] is probably the best example, and even this language is closer in spirit to a powerful macro assembler than to a high-level programming language.

2.4 Generality

A final requirement for languages that will support general-purpose programming is that they really be general purpose. This requirement is an obvious one, but it occasionally happens that generality is achieved at the expense of easy parallelization. Logic languages are a case in point. The problem is that many common operations cannot be implemented efficiently as logic programs. Examples include arithmetic, input and output, et cetera. The consequence of this fact is that all practical logic languages provide these features through extra-logical devices. For example, arithmetic in Prolog is implemented by interpreting certain symbols (e.g., "+") as denoting functions for which the run-time system has special evaluation rules (in contrast to most function symbols, which are wholly unevaluated). Input and output are generally implemented as predicates with side-effects, in violation of the usual view of predicates as side-effect free functions. The result of these deviations from pure logic programming is that parallelism that may be available based on the mathematical properties of pure logic is not always available in practical languages. For example, in a pure logic language the order in which terms in a conjunction are solved is irrelevant to the final solution — this property can be exploited as *and parallelism*. However, allowing predicates to have side-effects can make the order in which they are executed very important to the final meaning of a program. Similarly, expressions using evaluable function symbols can only be evaluated after the values of their inputs are known, again imposing an order on the computation. Needless to say, these and similar special cases make the exploitation of parallelism in real logic programs more difficult than it appears in theory. Note that the problem is not due as much to weaknesses in the basic ideas underlying logic programming as it is to severe limits on how thoroughly those ideas can be implemented at present. For example, arithmetic is treated specially in Prolog not because it is hard to imagine predicates that describe it, but because existing theorem proving algorithms cannot efficiently manipulate those predicates.

3 A Step Towards a Solution?

Of all the linguistic styles suggested for parallel programming, I find logic programming to be the most promising. Being declarative, logic languages can avoid the hard problems of data dependency analysis that plague imperative languages. As discussed above, logic languages also offer some intriguing possibilities for solving the problems that other declarative languages have with data parallelism. The one serious drawback with existing logic languages is the need to add extra-logical features in order to support really general-purpose programming. There is, however, a closely related class of languages called *constraint languages* that addresses this drawback. Like logic languages, constraint languages treat programs as systems of relations, and the goal of "executing" a program is to find values for unbound variables that make the relations hold. All of the advantages of logic languages for parallel programming appear to apply to constraint languages. Furthermore, where a logic language assumes no knowledge of what specific relations mean, a constraint language provides a set of primitive relations and data types about which compilers and interpreters do have considerable knowledge. More precisely, implementations of a constraint language must understand the algebraic properties of the primitives well enough to solve equations that use them. This extra knowledge means that one does not need to step outside the underlying mathematical formalisms of a constraint language in order to do general-purpose programming (at least in principle). Constraint satisfaction has been widely used as a programming technique (Sketchpad [42] and ThingLab [9] being two well-known examples), but only a few attempts have been made to base full programming languages on it [28, 41]. The authors of these languages did note that constraint languages seem appropriate for parallel programming, but did not seriously pursue the possibility.

The big problem with constraint languages is that it is very hard to find solutions to general constraints automatically. Like query evaluation for logic languages, constraint satisfaction can be viewed as theorem proving (prove that certain values for certain variables are a consequence of a system of constraints). However, the proofs that arise in constraint satisfaction are less structured than those that are encountered in logic languages. Thus the theorem provers that are used to implement logic languages are not powerful enough to implement constraint languages. The logic programming community is doing a great deal of promising research on more powerful theorem proving mechanisms [17, 25], but it remains unclear whether this work will be applicable to constraint languages. For now it appears that constraint language implementors will have to forego the elegant algorithmic implementations of logic languages in favor of more ad hoc heuristic ones. In return, one hopes, we will get languages that are both parallelizable and flexible.

I am testing the use of constraint languages for parallel programming at the University of Rochester. A prototype language called CONSUL has been developed, which I believe provides the flexibility necessary for general purpose programming and the sound formal foundations needed for automatic parallelism detection [5]. Current work on CONSUL involves writing a number of programs in it for a variety

of applications. These programs test the generality of the language (and point the way to extensions to make it more general), but more importantly will serve as inputs for a series of experiments aimed at estimating the parallelism that CONSUL really makes available. If these experiments bear out the expectation that CONSUL programs do exhibit substantial parallelism, later work will attempt to develop a parallelizing CONSUL compiler. A model for the parallel execution of CONSUL programs on which this compiler can be based has already been developed.

4 Summary and Conclusions

It has been noted before that software technology for parallel programming is in sad shape [26]. This paper extends beyond the earlier remarks by placing the blame squarely on the languages in which we are trying to write parallel programs, and even more fundamentally on the models of computation on which those languages are based. Two deep flaws in modern programming languages have been identified: One is the use of a side-effect based model of computation. This model is inherently sequential, and so anyone trying to find or express parallelism under it is seriously handicapped before they even begin. For this reason, imperative languages are ill suited for parallel programming. The second problem is an inability to express data parallelism without using iteration or recursion, both of which very effectively disguise parallelism as sequential computation. This problem is particularly severe in declarative languages, suggesting that only a small fraction of the parallelism in declarative programs will actually be recognizable.

The inadequacy of current programming languages is not the only reason why multiprocessors are hard to program. Parallel programming is also hindered by a dearth (at least relative to sequential programming) of highly parallel algorithms to be programmed. The importance of parallel algorithms cannot be ignored, and there are indeed many problems for which parallel algorithms are presently unknown. However, progress is being made on classifying problems that can be practically solved on parallel computers and in identifying algorithms for doing so [13, 22]. Much less progress is being made on devising practical languages for expressing these algorithms. Even the best parallel algorithms will not make multiprocessors or other parallel computers easy to use without languages in which those algorithms can be conveniently expressed.

One of the conclusions to be drawn from this work is that a number of common practices should be avoided by designers of languages for parallel programming. Among these are

- Reliance on side-effects.
- Use of sequential forms to describe data parallel computations.
- Partitioning into fine-grained processes with dense communication patterns.
- Sacrificing generality for parallelizability.
- Sharing of data between parts of a program.

The first three of these have already been discussed in detail. The fourth point was mentioned in connection with logic programming, where it was shown that

formalisms that are amenable to parallelization may need to be supplemented with other features that are not so easy to parallelize in order to make them suitable for general programming. The last point has not been discussed explicitly in this paper, but examples of problems stemming from it have been given. Among these are the need for global data dependency analysis (since data may be shared between widely separated parts of a program) and the difficulty of exploiting or-parallelism in logic languages (because the processes implementing different arms of an "or" may all try to bind the same shared variable).

A more important (and more positive) conclusion is that we must change the ways in which we think about programming in order to exploit the coming generations of massively parallel multiprocessors. Constraint languages and the CONSUL project represent a tentative step in this direction. Other steps are necessary, and the field is wide open for imaginative research. It is now easy to build parallel hardware; the next challenge is to develop programming models and languages that make this hardware easy to use.

Acknowledgements

Many thanks are due to Jerry Feldman, who encouraged me to write this paper in the first place and made many helpful comments on various drafts of it. Michael Scott and other colleagues at the University of Rochester also made helpful comments on early versions. Several members of the USENET Prolog community pointed out the translation of "forall" (and hence data parallelism) into or-parallel form.

Appendix I Implementations of "Totals"

This appendix presents implementations of "Totals" in several different programming languages. The languages include representatives of four of the linguistic classes discussed above — Pascal as a von Neumann language, Smalltalk for the object-oriented languages, an applicative subset of Lisp for the functional languages, and Prolog as a logic language. In addition to these, Lynx is used as an example of an explicitly parallel language.

Figure 4 shows a Pascal implementation of "Totals" Each input record is an instance of record type "*DataRec*", and the full set of records is represented by the array "*Data*". The totals and counts for each record type are also stored in arrays ("*Totals*" and "*Counts*"). The totals are computed by iterating over the *Data* array (lines 21–43), dispatching on the type of each record to a block of code that carries out the appropriate update to *Totals* (lines 23–42). The loop for computing totals also updates *Counts* as each record is encountered. Details of initializing the data records and printing the results have been left out, as have the exact sizes of arrays and the definitions of functions f and g. A complete program, from which Figure

4 is extracted, was tested under UNIX® 4.2 BSD on a VAX™ 11/750, using the "pc" compiler.

UNIX is a registered trademark of AT&T Bell Laboratories
VAX is a trademark of Digital Equipment Corporation

```
(1)  program Totals(input,output);
(2)      const
(3)          AuxMax = ...                          {Number of aux fields in a record}
(4)          SetSize = ...                         {Number of records to process}
(5)      type
(6)          DataRec= record
(7)                   RType : integer;
(8)                   Val   : real;
(9)                   Aux   : array [1..AuxMax] of real
(10)         end;
(11)     var
(12)        Data   : array [1..SetSize] of DataRec;
(13)        Totals : array [1..5] of real;
(14)        Counts: array [1..5] of integer;
(15)        i, j: integer;
(16)     begin
(17)        for i := 1 to 5 do begin
(18)           Counts[i] := 0;
(19)           writeln( 'Enter initial total for type ', i );   readln( Totals[i] )
(20)        end;
(21)        for i := 1 to SetSize do begin
(22)           counts[Data[i].RType] := Counts[Data[i].RType] + 1;
(23)           case Data[i].RType of
(24)                1: Totals[1] := Totals[1] + Data[i].Val;
(25)                2: begin
(26)                     if trunc(Data[i].Val/3.0) = Data[i].Val/3.0 then
(27)                       Totals[2] := Totals[2] + f(Data[i].Val)
(28)                     else
(29)                       Totals[2] := Totals[2] + g(Data[i].Val)
(30)                   end;
(31)                3: begin
(32)                     j := 1;
(33)                     while j <= AuxMax do begin
(34)                           Totals[3] := Totals[3] + Data[i].Aux[j];
(35)                           if (j < AuxMax - 1) and (j > 1) then
(36)                             Data[i].Aux[j - 1] := Data[i].Aux[j + 1]/2.0;
(37)                           j := j + 2
(38)                     end
(39)                   end;
(40)                4: Totals[4] := Totals[4] + ln(Data[i].Val);
(41)                5: Totals[5] := Totals[5] + f(Data[i].Val) + g(Data[i].Val)
(42)           end
(43)        end
(44)     end.
```

Figure 4. Algorithm "Totals" Implemented in Pascal

Figures 5a, 5b, and 5c show a Smalltalk implementation of "Totals". In keeping with the object-oriented paradigm, each of the major data items of the algorithm is implemented as a distinct object. Figure 5a describes the objects that represent individual data records, Figure 5b describes objects (only one of which is needed) that represent sets of records, and Figure 5c describes objects (again only one is needed) that represent records containing the counts and totals produced by the algorithm. The actual computations required by "Totals" are implemented as message-handling methods in the various object classes. Each data record (Figure 5a) has methods for accessing its various fields (lines 13–21), computing the sum of the odd-indexed "aux" elements (lines 22–27), and modifying the even-indexed "aux" elements (lines 29–32). Sets of records (Figure 5b) have a single method that serves as a driver for the totalling algorithm (lines 5–27). Result records (Figure 5c) have methods for initialization (lines 6–8 and 18–22) and for updating specific counts and totals (lines 11–16). As with all other presentations of "Totals", unspecified details like the sizes of sets or the exact nature of functions f and g have been left out of Figures 5a, 5b, and 5c. The actual program from which these figures are extracted was developed and run on a Sun™ workstation using Berkeley's "bs" Smalltalk system.

```
(1)  class name                    DataRec
(2)  superclass                    Object
(3)  instance variable names       type
(4)                                val
(5)                                aux
(6)  class variable names          AuxMax
(7)  class initialization
(8)     initialize
(9)        "Define size of 'aux' array in instances"
(10)       AuxMax ← ...
(11) instance methods
(12) Accessing
(13)    auxAt: i
(14)       "Return the i-th 'aux' field of receiver"
(15)       ↑ aux at: i
(16)    type
(17)       "Return the type field of receiver"
(18)       ↑ type
(19)    val
(20)       "Return the value field of receiver"
(21)       ↑ val
(22)    oddAuxes
(23)       "Returns the sum of odd-indexed 'aux' fields of receiver"
(24)       | sum |
(25)       sum ← 0.0.
(26)       (1 to: AuxMax by: 2) do: [ :i | sum ← sum + (aux at: i) ].
(27)       ↑ sum
(28) Modifying
(29)    updateEvenAuxes
(30)       "Sets each even-indexed 'aux' element of receiver to
(31)        1/2 the value of the next even-indexed 'aux' field"
(32)       (2 to: AuxMax−2 by: 2) do: [ :i | aux at: i put: (aux at: i+2) / 2.0 ]
```

Figure 5a. Smalltalk Implementation of Data Records for "Totals"

Sun is a trademark of Sun Microsystems, Inc.

```
(1)  class name                    RecordBag
(2)  superclass                    Bag
(3)  instance methods
(4)  Totalling
(5)     summarize: initialTotals
(6)        "Computes counts and totals by type for the records in the receiver."
(7)        | results |
(8)        results ← ResultRecord startingFrom: initialTotals.
(9)        self do: [ :datum |
(10)          results updateCountFor: datum type.
(11)          datum type = 1
(12)                ifTrue: [ results updateTotalFor: 1 by: datum val ].
(13)          datum type = 2
(14)                ifTrue: [ (datum type roundTo: 3) = datum type
(15)                             ifTrue: [ results updateTotalFor: 2
(16)                                                    by:datum val f ]
(17)                             ifFalse: [ results updateTotalFor: 2
(18)                                                    by:datum val g ] ].
(19)          datum type = 3
(20)                ifTrue: [ results updateTotalFor: 3 by: datum oddAuxes.
(21)                          datum updateEvenAuxes ].
(22)          datum type = 4
(23)                ifTrue: [ results updateTotalFor: 4 by: datum val ln ].
(24)          datum type = 5
(25)                ifTrue: [ results updateTotalFor: 5
(26)                                          by:datum val f + datum val g ]].
(27)       ↑ results
```

Figure 5b. Smalltalk Set of Records for "Totals"

```
(1)  class name                    ResultRecord
(2)  superclass                    Object
(3)  instance variable names       counts
(4)                                totals
(5)  instance creation
(6)     startingFrom: initialTotals
(7)        "Creates a new 'ResultRecord' with initial totals 'initialTotals'"
(8)        ↑ super new initialize: initialTotals
(9)  instance methods
(10) updating
(11)    updateCountFor: type
(12)       "Increments the count for type 'type' records by 1"
(13)       counts at: type put: (counts at: type) + 1
(14)    updateTotalFor: type by: amount
(15)       "Increments the total for type 'type' records by 'amount'"
(16)       totals at: type put: (totals at: type) + amount
(17) private
(18)    initialize: initialTotals
(19)       "Clears 'counts' field of receiver and sets 'totals' to 'initialTotals'"
(20)       counts ← #(0 0 0 0 0) shallowCopy.
(21)       totals ← initialTotals shallowCopy.
(22)       ↑ self
```

Figure 5c. Smalltalk Representation of Results from "Totals"

Figures 6a through 6g show "Totals" implemented in Lynx. This program consists of distinct processes to maintain the totals for each type of record, and a master process that distributes records to the totallers. This organization of the program was felt to be the best match between the parallelism in "Totals" and the process granularity for which Lynx was designed. Note that the even-indexed "aux" elements of type 3 records are updated by the master process (lines 18–27 and 44–46 of Figure 6b), even though one might expect this update to be part of the type 3 totalling process. This feature is a consequence of the message-passing model on which Lynx is based. Because processes do not share actual data structures, the only way a totalling process could return an updated record to the master would be to send it in a message. Keeping the update in the master process avoids the overhead of these additional messages for type 3 records. After the master process has distributed all of the data records to totalling processes, it collects the final totals by polling the totallers. There is a synchronization problem here, in that totallers must not report totals to the master until they have finished processing all the records sent to them. This synchronization is achieved by having each totaller count (in "pending") the number of records that it has received but not yet finished processing. Totals are reported to the master only when this count is zero. Figure 6c shows the complete implementation of this mechanism for Type 1 records. Enough synchronization of message senders with receivers and of entries within processes is built into Lynx that this simple approach works correctly. The other totalling processes are identical to the Type 1 totaller except for the *new_record* entries that perform the actual totalling. Because of the length of the complete Lynx program (over 200 lines), only these type-specific entries are shown for the other processes (Figures 6d, 6e, 6f, and 6g). The program from which these figures were extracted was developed and tested on a Butterfly multiprocessor, using the University of Rochester's Sun-based Lynx compiler.

```
(1)   header total_types;
(2)   const
(3)      AuxMax = ...;                          -- Number of aux fields in record
(4)   type
(5)      data_rec = record
(6)           rtype : integer;
(7)           val : real;
(8)           aux : array[1..AuxMax] of real;
(9)      end;
(10)  end total_types.
(11)  library total_types;
(12)  end total_types.
```

Figure 6a. Common Declarations for Lynx Version of "Totals"

```
(1)  process master;
(2)  use total_types;
(3)  const
(4)     SetSize = ...;                          -- Number of records to process
(5)  type
(6)     link_table = array[1..5] of link;
(7)     count_table = array[1..5] of integer;
(8)     data_table = array[1..SetSize] of data_rec;
(9)  var
(10)    links : link_table;
(11)    data : data_table;
(12)    counts : count_table;
(13)    initial_total : real;
(14)    cur_type : integer;
(15) entry init_total(total : real); remote;
(16) entry new_record(data : data_rec); remote;
(17) entry return_total : real; remote;
(18) procedure update_auxes(i : integer);
(19)    var
(20)       j : integer;
(21)    begin
(22)       j := 2;
(23)       while j < AuxMax - 2 do
(24)          data[i].aux[j] := data[i].aux[j + 2]/2.0;
(25)          j := j + 2;
(26)       end;
(27)    end update_auxes;
(28) begin
(29)    startprocess("type1_total", newlink(links[1]));
(30)    startprocess("type2_total", newlink(links[2]));
(31)    startprocess("type3_total", newlink(links[3]));
(32)    startprocess("type4_total", newlink(links[4]));
(33)    startprocess("type5_total", newlink(links[5]));
(34)    foreach i in [1..5] do
(35)       counts[i] := 0;
(36)       write("Initial total for type %d = ", i);
(37)       read("%f", initial_total);
(38)       connect init_total(initial_total|) on links[i];
(39)    end;
(40)    foreach i in [1..SetSize] do
(41)       cur_type := data[i].rtype;
(42)       counts[cur_type] := counts[cur_type] + 1;
(43)       connect new_record(data[i]|) on links[cur_type];
(44)       if cur_type = 3 then
(45)          update_auxes(i);
(46)       end;
(47)    end;
(48) end master.
```

Figure 6b. Master Process for Lynx Version of "Totals"

```
(1)   process type1_total(master : link);
(2)   use total_types;
(3)   var
(4)      pending : integer;
(5)      total : real;
(6)   entry init_total(init : real); remote;
(7)   entry new_record(data : data_rec);
(8)   begin
(9)      pending := pending + 1;
(10)     reply;
(11)     total := total + data.val;
(12)     pending := pending - 1;
(13)  end new_record;
(14)  entry return_total : real;
(15)  begin
(16)     await pending = 0;
(17)     reply(total);
(18)  end return_total;
(19)  begin
(20)     pending := 0;
(21)     accept init_total(total) on master;
(22)     reply;
(23)     bind master to new_record, return_total;
(24)  end type1_total.
```

Figure 6c. Totaller for Type 1 Records from Lynx Version of "Totals"

```
(1)   entry new_record(data : data_rec);
(2)   begin
(3)      pending := pending + 1;
(4)      reply;
(5)      if divisible_by_3(data.val) then
(6)           total := total + f(data.val);
(7)      else
(8)           total := total + g(data.val);
(9)      end;
(10)     pending := pending - 1;
(11)  end new_record;
```

Figure 6d. Core of Lynx Type 2 Record Totaller

```
(1)   entry new_record(data : data_rec);
(2)   var
(3)      i : integer;
(4)   begin
(5)      pending := pending + 1;
(6)      reply;
(7)      i := 1;
(8)      while i <= AuxMax do
(9)           total := total + data.aux[i];
(10)          i := i + 2;
(11)     end;
(12)     pending := pending - 1;
(13)  end new_record;
```

Figure 6e. Core of Lynx Type 3 Record Totaller

```
(1)   entry new_record(data : data_rec);
(2)   begin
(3)      pending := pending + 1;
(4)      reply;
(5)      total := total + ln(data.val);
(6)      pending := pending - 1;
(7)   end new_record;
```

Figure 6f. Core of Lynx Type 4 Record Totaller

```
(1)   entry new_record(data : data_rec);
(2)   begin
(3)      pending := pending + 1;
(4)      reply;
(5)      total := total + f(data.val) + g(data.val);
(6)      pending := pending - 1;
(7)   end new_record;
```

Figure 6g. Core of Lynx Type 5 Record Totaller

Figure 7 shows "Totals" implemented in Prolog. The main predicate in this program is "totals/5"[6] (lines 1–6). This predicate is intended to be invoked by users with its last two arguments bound to the set of records to be totalled and the initial totals, respectively. Upon return, the first three arguments will be bound to the totals, the counts, and an updated set of records (to reflect the changes to the "aux" elements in type 3 records). Note how "totals/5" recursively steps through the set of records, removing one record on each pass for processing by "one_rec/4". This is a good example of recursion replacing iteration in a declarative language. "One_rec/4" (lines 7–25) is the heart of the program, being the predicate that dispatches on record type to do the actual totalling. Prolog's unification mechanism is used to do the actual testing of record types, an approach to conditionals that is typical of logic programming. Predicates "sum/2" (lines 26–28) and "halve/2" (lines 29 33) perform the summation of odd-indexed "aux" elements and the modification of even-indexed ones. This code is taken from a program that was written and tested using C-Prolog under UNIX 4.3 BSD on a VAX 11/750.

```
(1)   totals( Ts, Cs, [NewR|NewRs], [R|Rs], InitTs ) :-
(2)       one_rec( Ts1, Cs1, NewR, R ),
(3)       totals( Ts2, Cs2, NewRs, Rs, InitTs ),
(4)       list_sum( Ts, Ts1, Ts2 ),
(5)       list_sum( Cs, Cs1, Cs2 ).
(6)   totals( InitTs, [0,0,0,0,0], [], [], InitTs ).

(7)   one_rec( [V,0,0,0,0], [1,0,0,0,0], [1,V,Aux], [1,V,Aux] ).
(8)   one_rec( [0,X,0,0,0], [0,1,0,0,0], [2,V,Aux], [2,V,Aux] ) :-
(9)       D1 is V/3.0,
(10)      D2 is floor(D1),
(11)      D1 == D2,
(12)      f(X,V).
(13)  one_rec( [0,X,0,0,0], [0,1,0,0,0], [2,V,Aux], [2,V,Aux] ) :-
(14)      D1 is V/3.0,
(15)      D2 is floor(D1),
(16)      D1 \== D2,
(17)      g(X,V).
(18)  one_rec( [0,0,X,0,0], [0,0,1,0,0], [3,V,Aux2], [3,V,Aux] ) :-
(19)      sum(X,Aux),
(20)      halve(Aux2,Aux).
(21)  one_rec( [0,0,0,X,0], [0,0,0,1,0], [4,V,Aux], [4,V,Aux] ) :- X is log(V).
(22)  one_rec( [0,0,0,0,X], [0,0,0,0,1], [5,V,Aux], [5,V,Aux] ) :-
(23)      f(X1,V),
(24)      g(X2,V),
(25)      X is X1+X2.

(26)  sum( X, [N1,N2|Ns] ) :- sum( Y, Ns ), X is Y+N1.
(27)  sum( N, [N] ).
(28)  sum( 0, [] ).

(29)  halve( [X,H|NewAuxes], [X,Y,Z,D|Auxes] ) :- H is D/2, halve(NewAuxes,[Z,D|Auxes]).
(30)  halve( [X,Y,Z], [X,Y,Z] ).
(31)  halve( [X,Y], [X,Y] ).
(32)  halve( [X], [X] ).
(33)  halve( [], [] ).

(34)  list_sum( [L|Ls], [L1|Ls1], [L2|Ls2] ) :-
(35)      L is L1+L2,
(36)      list_sum( Ls, Ls1, Ls2 ).
(37)  list_sum( [], [], [] ).
```

Figure 7. "Totals" Implemented in Prolog

[6] It is common when discussing Prolog programs to identify predicates by name and number of arguments

Figure 8 shows an implementation of "Totals" in an applicative subset of Common Lisp. The main function in this program is "Total" (lines 3–9), which takes as arguments a list of records and a list of initial totals, and returns a record (of type "Stats", see line 1) containing the summary statistics (totals, counts, and modified records) required by "Totals". Returning a record in this manner is one way of implementing operations that logically produce several results as functions with only a single return value.[7] "Total" is organized as a recursive traversal of the list of records, with one record being removed for processing at each step. Note that the recursive call returns a base result which is then extended to include the current record by function "Process-Record" (lines 10–38). "Process-Record" is just a big "case" statement that dispatches to the appropriate processing based on record type. Logically "Process-Record" receives a set of summary statistics in "Base-Stats", which it updates to include "Record". Because functional languages do not allow side-effects however, "Base-Stats" cannot be updated directly — instead, "Process-Record" must build a new "Stats" record, derived from "Base-Stats", each time it is called. The key function for building these new records is "Incr-Element" (lines 49–58), which adds a specified increment to a specified element of a list, returning a new list that is identical to the original in all positions except the one incremented. The program from which this code is taken was written in Common Lisp and run on a Texas Instruments Explorer™ workstation.

[7] Many functional languages, including Common Lisp, allow functions to have multiple return values, but the mechanisms for doing so are awkward.

Explorer is a trademark of Texas Instruments, Inc.

```
(1)  (defstruct Stats     Totals Counts Records)
(2)  (defstruct Datum     Type Val A  x)
(3)  (defun Total (Records Initial-Totals)
(4)     (cond
(5)       ((null Records)   (Make-Stats :Totals Initial-Totals
(6)                                     :Counts (list 0 0 0 0 0)
(7)                                     :Records '() ))
(8)       (t   (Process-Record (car Records)
(9)                            (Total (cdr Records) Initial-Totals)))))
(10) (defun Process-Record (Rec Base-Stats)
(11)    (case (Datum-Type Rec)
(12)          ( (1)   (Make-Stats :Totals (Incr-Element (Stats-Totals Base-Stats) 1
(13)                                                    (Datum-Val Rec))
(14)                              :Counts (Incr-Element (Stats-Counts Base-Stats) 1 1)
(15)                              :Records (cons Rec (Stats-Records Base-Stats))))
(16)          ( (2)   (Make-Stats :Totals (Incr-Element (Stats-Totals Base-Stats) 2
(17)                                                    (if (= (* (truncate (Datum-Val Rec) 3) 3)
(18)                                                           (Datum-Val Rec))
(19)                                                       (f (Datum-Val Rec))
(20)                                                       (g (Datum-Val Rec))))
(21)                              :Counts (Incr-Element (Stats-Counts Base-Stats) 2 1)
(22)                              :Records (cons Rec (Stats-Records Base-Stats))))
(23)          ( (3)   (Make-Stats :Totals (Incr-Element (Stats-Totals Base-Stats) 3
(24)                                                    (Sum-Odds (Datum-Aux Rec)))
(25)                              :Counts (Incr-Element (Stats-Counts Base-Stats) 3 1)
(26)                              :Records (cons (Make-Datum :Type (Datum-Type Rec)
(27)                                                         :Val (Datum-Val Rec)
(28)                                                         :Aux (Halve-Evens (Datum-Aux Rec)))
(29)                                             (Stats-Records Base-Stats))))
(30)          ( (4)   (Make-Stats :Totals (Incr-Element (Stats-Totals Base-Stats) 4
(31)                                                    (log (Datum-Val Rec)))
(32)                              :Counts (Incr-Element (Stats-Counts Base-Stats) 4 1)
(33)                              :Records (cons Rec (Stats-Records Base-Stats))))
(34)          ( (5)   (Make-Stats :Totals (Incr-Element (Stats-Totals Base-Stats) 5
(35)                                                    (+ (f (Datum-Val Rec))
(36)                                                       (g (Datum-Val Rec))))
(37)                              :Counts (Incr-Element (Stats-Counts Base-Stats) 5 1)
(38)                              :Records (cons Rec (Stats-Records Base-Stats))))))
(39) (defun Sum-Odds (L)
(40)    (case (length L)
(41)          ((0)   0)
(42)          ((1)   (car L))
(43)          (otherwise   (+ (car L) (Sum-Odds (cddr L))))))
(44) (defun Halve-Evens (L)
(45)    (cond
(46)     ((< (length L) 4)   L)
(47)     (t   (cons (nth 0 L)
(48)                (cons (/ (nth 3 L) 2) (Halve-Evens (cddr L)))))))
(49) (defun Incr-Element (Base-List I Incr)
(50)    (labels ( (Make-Copy (L Pos)
(51)                 (cond
(52)                  ((null L)   '() )
(53)                  (t   (if (= Pos I)
(54)                           (cons (+ (car L) Incr)
(55)                                 (Make-Copy (cdr L) (+ Pos 1)))
(56)                           (cons (car L)
(57)                                 (Make-Copy (cdr L) (+ Pos 1))))))) )
(58)       (Make-Copy Base-List 1)))
```

Figure 8. "Totals" Implemented in Lisp

References

[1] W. Ackerman. "Data Flow Languages". *Computer*, Feb. 1982. pp. 15-23.

[2] G. Andrews. "The Distributed Programming Language SR - Mechanisms, Design, and Implementation". *Software — Practice and Experience*, Aug. 1982 (**12:8**). pp. 719-753.

[3] "Butterfly™ Parallel Processor Overview". BBN Laboratories Inc., June 1985.

[4] J. Backus. "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". *Communications of the ACM*, Aug. 1978 (**21:8**). pp. 613-641.

[5] D. Baldwin and C. Quiroz. "Parallel Programming and the CONSUL Language". Proceedings of the 1987 International Conference on Parallel Processing.

[6] U. Banerjee. "Speedup of Ordinary Programs". Technical Report number UIUCDCS-R-79-989, Department of Computer Science, University of Illinois at Urbana-Champaign, Oct. 1979.

[7] A. Black *et al.* "Distribution and Abstract Types in Emerald". *IEEE Transactions on Software Engineering*, Jan. 1987 (**SE-13:1**). pp. 65-76.

[8] G. Blelloch. *AFL-1: A Programming Language for Massively Concurrent Computers.* M.S. Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1986.

[9] A. Borning. "The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory". *ACM Transactions on Programming Languages and Systems*, Oct. 1981 (**3:4**). pp 353-387.

[10] B. Brode. "Precompilation of FORTRAN Programs to Facilitate Array Processing". *Computer*, Sept. 1981 (**14:9**). pp. 46-51.

[11] K. Clark and S. Gregory. "PARLOG: Parallel Programming in Logic". *ACM Transactions on Programming Languages and Systems*, Jan. 1986 (**8:1**). pp. 1-49.

[12] W. Clocksin and C. Mellish. *Programming in Prolog.* Berlin: Springer-Verlag, 1981.

[13] S. Cook. "A Taxonomy of Problems with Fast Parallel Algorithms". *Information and Control*, 1985 (Vol. 64). pp. 2-22.

[14] J. Ellis. *Bulldog: A Compiler for VLIW Architectures.* Ph. D. Dissertation, Department of Computer Science, Yale University, Feb. 1985.

[15] J. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction". *IEEE Transactions on Computers*, July 1981 (**C-30:7**). pp. 478-490.

[16] K. Frenkel. "Evaluating Two Massively Parallel Machines". *Communications of the ACM*, Aug. 1986 (**29:8**). pp. 752-758.

[17] J. Gallier and S. Raatz. "SLD-Resolution Methods for Horn Clauses with Equality Based on E-Unification". Proceedings of the 1986 Symposium on Logic Programming, Salt Lake City, Utah (IEEE). pp. 168-179.

[18] D. Gelernter. "Domesticating Parallelism" (Guest Editor's Introduction). *Computer*, Aug. 1986 (**19:8**). pp. 12-16.

[19] D. Gelernter. "Generative Communication in Linda". *ACM Transactions on Programming Languages and Systems*, Jan. 1985 (**7:1**). pp. 80-112.

[20] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Reading, Ma.: Addison-Wesley, 1983.

[21] W. Hillis. *The Connection Machine*. Cambridge, Ma: MIT Press, 1985.

[22] W. Hillis and G. Steele. "Data Parallel Algorithms". *Communications of the ACM*, Dec. 1986 (**29:12**). pp. 1170-1183.

[23] C. Hoare. "Communicating Sequential Processes". *Communications of the ACM*, Aug. 1978 (**21:8**). pp. 666-677.

[24] K. Jensen and N. Wirth. *Pascal User Manual and Report*. Berlin: Springer-Verlag, 1974.

[25] A. Josephson and N. Dershowitz. "An Implementation of Narrowing: The RITE Way". Proceedings of the 1986 Symposium on Logic Programming, Salt Lake City, Utah (IEEE). pp. 187-197.

[26] A. Karp. "Programming for Parallelism". *Computer*, May 1987 (**20:5**). pp. 43-57.

[27] B. Liskov and R. Scheifler. "Guardians and Actions: Support for Robust, Distributed Programs". *ACM Transactions on Programming Languages and Systems*, July 1983 (**5:3**). pp. 381-404.

[28] J. Maleki. *ICONStraint, A Dependency Directed Constraint Maintenance System*. Licentiate Thesis number 71, Dept. of Computer and Information Science, Linköping University, 1987.

[29] J. McGraw. "The VAL Language: Description and Analysis". *ACM Transactions on Programming Languages and Systems*, Jan. 1982 (**4:1**). pp. 44-82.

[30] M. Metcalfe. "FORTRAN 8X — The Emerging Standard". *FORTRAN Forum*, April 1987 (**6:1**). pp. 28-47.

[31] A. Nicolau. *Parallelism, Memory Anti-Aliasing and Correctness for Trace-Scheduling Compilers*. Ph. D. Dissertation, Department of Computer Science, Yale University, Mar. 1985.

[32] D. Padua and M. Wolfe. "Advanced Compiler Optimizations for Supercomputers". *Communications of the ACM*, Dec. 1986 (**29:12**). pp. 1184-1201.

[33] R. Perrott and A. Zarea-Aliabadi. "Supercomputer Languages". *ACM Computing Surveys*, Mar. 1986 (**18:1**). pp. 5-22.

[34] R. Polivka and S. Pakin. *APL: The Language and Its Usage*. Englewood Cliffs, N.J: Prentice-Hall, 1975.

[35] D. Rumelhart and J. McClelland eds. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. I *Foundations*. Cambridge, Ma: MIT Press, 1986.

[36] M. Scott. "Language Support for Loosely-Coupled Distributed Programs". *IEEE Transactions on Software Engineering*, Jan. 1987 (**SE-13:1**). pp. 88-103.

[37] C. Seitz. "The Cosmic Cube". *Communications of the ACM*, Jan. 1985 (**28:1**). pp. 22-33.

[38] E. Shapiro. "Concurrent Prolog: A Progress Report". *Computer*, Aug. 1986 (**19:8**). pp. 44-58.

[39] B. Smith. "A Pipelined, Shared Resource MIMD Computer". Proceedings of the 1978 International Conference on Parallel Programming, Aug. 1978. pp. 6-8.

[40] G. Steele. *Common Lisp: The Language*. Digital Press, 1984.

[41] G. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints*. Ph. D. Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Aug. 1980.

[42] I. Sutherland. "SKETCHPAD: A Man-Machine Graphical Communication System". Technical Report number 296, Massachusetts Institute of Technology Lincoln Laboratory, Jan. 1963.

END

DATE

3-88

DTIC